riacs Research Institute for Advanced Computer Science

Simulation-Based Verification of a Propulsion Feed Subsystem via Livingstone PathFinder

Tony Lindsey

Charles Pecheur

RIACS Technical Report 03.11

September 2003

Simulation-Based Verification of a Propulsion Feed Subsystem via Livingstone PathFinder

Tony Lindsey, QSS Group

Charles Pecheur, RIACS

RIACS Technical Report 03.11

September 2003

Livingstone PathFinder (LPF) is a software tool for automatically analyzing model-based diagnosis applications across a wide range of scenarios. This technical report describes experimental results from the application of LPF to a Livingstone-Based diagnosis for the propulsion feed subsystem of the X-34 space vehicle. We describe Livingstone, LPF and the X-34 application, report statistics on LPF performance, including improvements through the use of heuristic-based guided search, and analyze two examples of errors reported by LPF.

This work was supported in part by the National Aeronautics and Space Administration under Cooperative Agreement NCC 2-1006 with the Universities Space Research Association (USRA), as well as CSRDS Task 091 with QSS Group, Inc. The work was performed in the context of NASA ECS Project 2.2.1.1 "Validation of Model-Based IVHM Architectures".

Contents

1 Introduction

Livingstone PathFinder (LPF) is a simulation-based verification tool that drives a Livingstone Diagnosis system, embedded in a simulated environment, across a wide range of command and fault scenarios. The environment consists of a *Driver* that enumerates the commands and faults, and a *Simulator* that simulates the response of the system being diagnosed. Check-pointing is implemented in the Diagnosis engine, the Simulator and the Driver, to allow backtracking and exploration of alternate scenarios. Currently, a second instance of a Livingstone engine is used for the Simulator.

LPF is intended to offer a variety of state space exploration strategies, along with a flexible way of changing to the desirable techniques. Most recently a best-first (heuristic) search has been implemented and added to the list of offered strategies.

This report describes experiments in applying LPF to a real-size Livingstone-based application. The PITEX project is applying Livingstone to the propulsion feed subsystem of the X-34 space vehicle. We have used the X-34 Livingstone model to test, evaluate and compare the different features of LPF, in particular the heuristic search capabilities.

The report describes these experiments, reports execution statistics, and discusses some of the error conditions that were found.

2 Livingstone

Livingstone is a model-based diagnosis system that uses a qualitative model of the different components of a physical plant and their interactions, both under nominal and faulty conditions [4]. It tracks the commands issued to the physical system, then compares the state predicted by the model against observations received from physical sensors.

If a discrepancy is detected, Livingstone performs a diagnosis by searching for combinations of faults that are consistent with the observations. Each combination is called a *candidate* and has an associated *rank* estimating its probability, higher ranked candidates being less likely. Livingstone's best-first search algorithm returns more likely, lower ranked candidates first. In particular, when the nominal case is consistent with Livingstone's observations, the empty candidate (of rank 0) is generated.

The Livingstone fault diagnosis and recovery kernel has successfully been applied to the Remote Agent Experiment (RAX) demonstration on Deep Space 1 (DS-1) in 1999. A new generation of Livingstone, called L2, includes temporal trajectory tracking. Further extensions supporting more general constraint types (hybrid discrete/continuous models) are under investigation.

The L2 engine is avaiable as a compiled library (L2 is written in C++). A utility program, `l2test`, provides a command line interface to perform all L2 operations interactively. `l2test` commands can be stored in *L2 scenario* files for batch replay.

3 Livingstone PathFinder

Livingstone PathFinder (LPF) is a simulation-based tool for analyzing and verifying Livingstone-based diagnosis applications. LPF executes a Livingstone diagnosis engine, embedded into a simulated environment, and runs that assembly through all executions described by a user-provided scenario file, while checking for various selectable error conditions after each step.

The architecture of LPF is depicted in Figure 1. The tool is built in a modular way with generic interfaces, to allow easy substitution of alternative versions of its different parts. Altogether, this provides a flexible, extensible framework for simulation-based analysis of diagnosis applications.

The testbed under analysis consists of the following three components:

- *Diagnosis*: the diagnosis system being analyzed. LPF currently supports the Livingstone engine; extension to the Titan system [1] is under study.
- *Simulator*: the simulator for the device on which diagnosis is performed. Currently a second Livingstone engine instance is used for the simulator module, but the architecture is open to other alternatives.
- *Driver*: the simulation driver that generates commands and faults according to a user-provided scenario file.

All three components are instrumented so that their execution can be single-stepped in both forward and backward directions. The *Search Engine* controls the order in which the resulting graph of possible executions is explored.

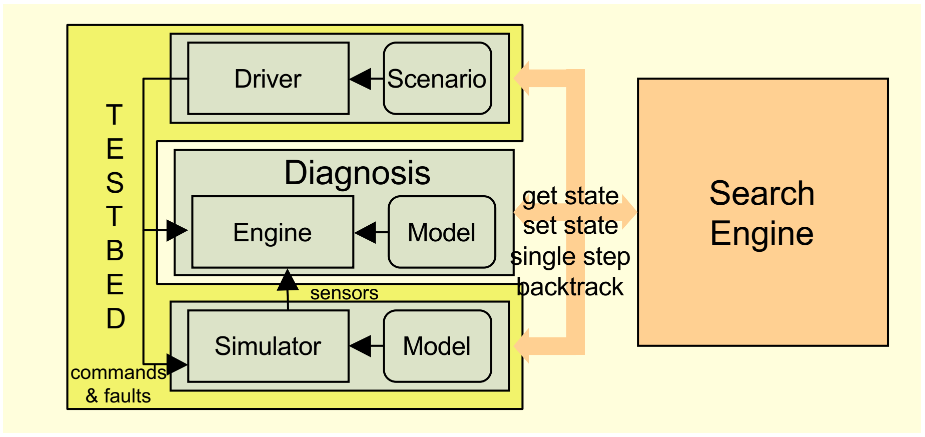

Figure 1: Livingstone PathFinder Architecture

The scenario file is essentially a non-deterministic program whose elementary instructions are commands and faults. Scenarios are built from individual instructions using sequential, concurrent (interleaved) and choice statements. The sample scenario in Figure 2 defines a sequence of three commands `cmd1`, `cmd2` and `cmd3`, with one fault chosen among `fltA` and `fltB` occurring at some point in the sequence. LPF provides a way to automatically generate a scenario combining all commands and faults of a model following the same sequence/choice pattern.

```
mix {
  "command cmd1";
  "command cmd2";
  "command cmd3";
} and {
  choose "fault fltA";
  or "fault fltB";
}
```

Figure 2: Sample LPF scenario

Each event produced by the Driver is passed to the Simulator, which updates its state accordingly. Commands (but not faults) are also passed to Diagnosis. Updated observable variables are then extracted from the Simulator and passed on to Diagnosis. This cycle is repeated along all sequences covered by the scenario, saving and restoring intermediate states to explore alternate routes. User-selectable properties, such as consistency between the diagnosis results and the actual state of the Simulator, are checked at each step, and a trace is reported if a violation is detected.

The latest version of LPF offers a number of useful auxiliary capabilities, such as generating an expanded scenario tree, counting scenario states, exploring the simulator alone (no diagnosis), generating a default scenario, and producing a wide array of diagnostic and debugging information, including traces that can be replayed in Livingstone simulation tools.

3.1 LPF Error Conditions

In each state along its exploration, LPF can check one or several *error conditions* among a user-selectable set. Currently, LPF supports the following error conditions:

- *Simulator consistency*: Simulator reaches an inconsistent state, typically occurring after executing a command or injecting a fault that results in conflicts among the model constraints and assignments.

- *Diagnosis consistency*: Diagnosis reaches an inconsistent state, after failing to find any candidate consistent with previous commands and observations. This condition could also be found by using Stanley, a graphical user interface to Livingstone, to test scenarios. However LPF has the advantage of more exhaustive automated test coverage. This particular error condition is unlikely given the “unknown” mode usage. The “unknown” failure mode represents a catchall mode for unspecified faults.

- *Mode comparison (MC)*: compares the modes of all components in the Simulator to those assumed by Diagnosis, and reports any discrepancy.

- *Candidate matching (CM)*: checks that at least one Diagnosis candidate matches (i.e. has the same faults as) the Simulator state.

- *Candidate subsumption (CS)*: checks that at least one Diagnosis candidate is subsumed by (i.e. has its faults included in) the Simulator state.

While the first two conditions can prove to be useful debugging tools, the subsequent conditions address the core of diagnosis correctness at three different levels of generality, from the most restrictive to the least. They constitute three successive refinement steps based on the intuition that diagnosis should properly track the state of the (simulated) system. *Mode comparison* only considers the most likely candidate and reports errors even if another reported candidate matches the state, which is overly restrictive in practice since the injected fault may not be the most likely one. Instead, the *Candidate matching* condition considers all candidates. Even then, a fault may remain unobservable without causing immediate harm as long as its component is not solicited. Experience reveals that this is a frequent situation, causing a large proportion of spurious error reports. In contrast, *Candidate subsumption* only reports cases where none of the diagnosed fault sets is *included* in the actual fault set. For example, the empty candidate is subsumed by any fault set and thus never produces an error using this condition. While this will not detect cases where Diagnosis misses harmful faults, it will catch cases where the wrong faults are reported by the Diagnosis engine. This condition has proven to be the most productive so far, as further discussed in Section 5.

3.2 Search Strategies

LPF supports alternative state space exploration strategies. The first implemented search strategy is a straightforward depth-first search; and an implementation of *heuristic search* using configurable fitness functions has just been completed. Other strategies, such as interactive (i.e. user-driven) or randomized search, are under consideration.

Heuristic search uses a fitness function to rank different states according to a given criterion and explores them in that order. Provided that a good heuristic can be found, i.e. one that favors states that are closer to errors, heuristic search offers two advantages: The first is that errors are likely to be found earlier on and the second is the production of shorter counterexample paths. So far we have implemented and experimented with the two following heuristics:

- *Breadth-First Search* (BFS) uses the execution depth, i.e. the number of steps to that state. This heuristic finds the shortest counter-examples. Although this exploration strategy does not search the state space intelligently it was effective at detecting *Mode comparison* error condition violations which are the most frequently occurring violation.

- *Candidate Count* (CC) uses the number of generated candidates obtained via the diagnosis system. For any particular state this number is not constant and has a configurable upper bound. Intuitively, this heuristic leads to states with fewer candidates and thus is more likely to lead to cases where no correct diagnoses (or even no diagnosis at all) is generated.

3.3 Verification Using LPF

Because it executes the actual Livingstone system, LPF is a fairly general verification tool for Livingstone applications, in that it can reveal the following different types of problems:

- *Engine errors*, i.e. inaccurate diagnosis due to errors in the diagnosis program. These are not the main concern, as the engine is re-used across many applications and therefore presumably more mature and stable.
- *Diagnosability errors*, i.e. inaccurate diagnosis due to lack of observability of the system's internal state.
- *Incompleteness errors*, i.e. inaccurate diagnosis due to incomplete search (e.g. resulting from limited memory resources or erroneous Livingstone parameter settings).
- *Modeling errors*, i.e. inaccurate diagnosis due to aspects of the system incorrectly or too coarsely represented in the Livingstone model. This a major issue, as Livingstone operates on highly abstracted models of the physical system.
- *Integration errors*, where the diagnosis system fails to properly interact with its simulated environment, and the simulator in particular.

The last two items become relevant only when a higher-fidelity simulation of the diagnosed system is used. Alternatively, using the same Livingstone model for the simulator focuses exclusively on engine, diagnosability and incompleteness errors.

3.4 Simulators in LPF

The modular design of LPF allows the use of different simulators through a generic application programming interface (API). As a first step, we have been using a second Livingstone engine instance for the Simulator (used in a different way: simulation infers outputs from inputs and injected faults, whereas diagnosis infers faults given inputs and outputs).

Using the same model for simulation and diagnosis, as this implies, may appear as circular reasoning but has its own merits. It provides a methodological separation of concerns: by doing so, we validate operation of the diagnostic system under the assumption that the diagnosis model is a perfect model of the physical system, thus concentrating on proper operation of the diagnosis algorithm itself. Incidentally, it also provides a cheap and easy way to set up a verification testbed, even in the absence of an independent simulator.

On the other hand, using the same model for the simulation ignores the issues due to inaccuracies of the diagnosis model wrt the physical system, which is a main source of problems in developing model-based applications. We are currently considering integration of higher fidelity simulators (e.g. Matlab or Simulink models). We have already been studying integration of NASA Johnson Space Center's CONFIG simulator system [2]. Note that higher fidelity simulators are likely to be less flexible and efficient; for example, they may not have backtracking or check-pointing.

4 X-34 System Model

The Livingstone diagnosis system is being considered for Integrated Vehicle Health Maintenance (IVHM) for NASA's next-generation space vehicles. In that context, the PITEX experiment has demonstrated the application of Livingstone-based diagnosis to the main propulsion feed subsystem of the X-34 space vehicle [5, 3], and LPF has been successfully applied to the PITEX model of X-34. The main propulsion system model components are specified by nominal and fault modes. The full model contains 535 components and 823 attributes, 250 transitions, compiling to 2022 propositional clauses.

The model consists of the following four subsystems[5]:

- *Pressurization*: provides pressurized helium to the propellant tanks during the bleed and burn phases.
- *Pneumatics*: provides pressurized helium to open and close pneumatic valves.
- *Liquid Oxygen (LOX)*: has two tanks for storing liquid oxygen: a forward and aft tank. All of the logic is implemented in the forward tanks, and the aft tank is a simple pass-through component.
- *Rocket Propellent (RP-1)*: subsystem consisting of one RP-1 tank.

In addition, two different scenarios have been used to analyze the X-34 system model:

- The *PITEX baseline scenario* combines one nominal and 29 failure scenarios, derived from those used by the PITEX team for testing Livingstone, as documented in [3]. This scenario covers 89 states.
- The *random scenario* covers a set of commands and faults combined according to the sequence/choice pattern of Figure 2. This scenario is an abbreviated version of the automatically generated LPF scenario, and covers 10216 states.

5 Experimental Results

The LPF software tool covered the *PITEX baseline* and *random* scenario at an average rate of 50 to 100 states per minute. Early experiments demonstrated the technical viability of the approach and provided informative feedback to the developers. Experience revealed a need for improved post-treatment of generated results, to filter critical information from large amounts of generated data. Our experiments involved the usage of both depth-first and heuristic search.

LPF provides a number of options for adjusting analysis parameters and generating results and monitoring information in various forms. In our experiments, we typically used the following set of options:

- `+cmdlogtrace` – For each error found, create L2 scenario files (both for simulation and diagnosis) reproducing the sequence leading to that error. These files can be replayed in `l2test` for further analysis (see Section 2).
- `+debug` – Print additional monitoring data for debugging purposes.
- `+log` – Create L2 scenario files (both for simulation and diagnosis) corresponding to the complete execution, including backtracking between different traces.
- `+trace` – Print a trace of the current execution whenever an error occurs.
- `+verbose` – Report the transition performed and a summary of resulting changes at each step.

We will first discuss scalability and compare different error conditions using depth-first search, then discuss the benefits of heuristic search and finally illustrate some of the errors found.

5.1 Depth-First Search

Our initial experiments involved searching the entire state space. The results use the *search all* termination criterion which specifies that the search proceed normally after an error is detected and report any additionally found errors. We used the depth-first algorithm in this case, since heuristic search offers no advantage when searching through all states of the scenario. Table 1 summarizes depth-first search statistics.

When searching for either *Mode comparison* or *Candidate matching* LPF reported an excessive number of errors, most of which were *trivial*, in the following sense. When a fault produces no immediately observable effect, then Livingstone reports does not infer any abnormal behaviour and reports the empty candidate. For example, although a valve may be stuck, the fault will stay unnoticed until that valve is acted on. While these errors are indeed missed diagnoses, experience shows that they are in most cases expected and do not constitute significant diagnosis flaws. In particular, searching for *Candidate*

Scenario	Search	Error Cond.	#Errors	#Non-trivial	States	States/min
Baseline	all	CM	27	4	89	44
Baseline	all	CS	0	0	89	67
Random	all	CM	9621	137	10216	51
Random	all	CS	5	5	10216	52
Random	one	CS	1	1	8648	49
Random	min	CS	2	2	8838	44

Table 1: Depth-first search statistics

matching error condition violations the *PITEX baseline* scenario produced 27 errors of which only 4 were not trivial. The identical search was performed over the *random* scenario resulting in 9621 errors of which 137 were not trivial.

In contrast, verification over the *random* scenario using the *Candidate subsumption* (CS) error condition reported a total of 5 errors, none of which were trivial (the *PITEX* baseline scenario reported no errors). Indeed, the CS error condition will only trigger if all candidates report some incorrect fault, but not when faults are missed. In particular, the empty candidate subsumes all cases and will, by definition, never trigger a CS error condition violation.

For illustration purposes, the two last rows in Table 1 show results using alternative termination criteria *search-one* (stop at first error) and *search-min* (keep searching for shorter error traces only). Search-one finds the first error (at depth 16) only after 8648 states, i.e. after it has covered a large part of the state space already. Because of this, the benefit of using search-min is rather marginal in this case; search-min finds a second error, at depth 3, by exploring only 190 additional states.

An example of actual statistics generated by LPF is listed below. This results from a depth-first search on the Random scenario, searching all errors. The options were as follows:

```
-mode full -search all -cbfs 5 100 100 3 5 -check sim -check mir
-check modes +check subsumption -completion +verbose +debug +trace
+cmdlogtrace +log MainPropulsionSystem
```

After generating monitoring and error trace data (in large amounts), LPF reports final execution statistics as quoted below. Numbers are provided both for the top-level execution and for the Simulator and Diagnosis components; the information is mostly redundant and useful for tool debugging purposes only.

```
Statistics
==========
  10216 states
  16 max depth
  5 errors
Simulator:
  10214 commands
  10215 completions
  15 max depth
Diagnosis:
  10214 commands
```

Strategy	Max depth	Error Cond	Time	States	States/min
DFS	16	CS	02:55:38	8648	49
BFS	3	CS	00:03:56	154	38
CC	5	CS	00:03:42	154	38
DFS	16	CM	00:00:15	17	68
BFS	2	CM	00:00:13	4	20
CC	1	CM	00:00:11	4	24

Table 2: Comparison of heuristic vs. depth-first search

```
  10215 observable updates
  274 diagnoses of inconsistent states
  15 max depth

Total time 3:14:13
Number of states per minute 52
```

5.2 Heuristic Search

In this section, we illustrate the benefits of using heuristic search. The experiments used the Random scenario and the *search-one* option, i.e. LPF stopped at the first error found. The fitness function types used were *breadth-first search* (BFS) and *candidate count* (CC) (see Section 3.2). The state space was searched for the *Candidate subsumption* (CS) and *Candidate matching* (CM) error condition violations. The results are summarized in Table 2, including depth-first search (DFS) for comparison purposes.

Results show that, using BFS, LPF mapped through only 154 states before detecting a CS error condition violation at depth 3, compared to 8648 states to depth 16 using depth-first search. CC also explores 154 states, although it goes a little deeper to depth 5. For the less selective CM error condition, LPF finds an error after just 4 states using CC or BFS, versus 17 states with DFS. As expected, heuristic search (both BFS and CC) detected the error significantly faster than depth-first search. These results illustrate that heuristic search can save a great deal of time by skipping large parts of the search space where errors are less likely to be found.

5.3 Example Scenario 1: SV31 double fault

Our first example involves the *PITEX baseline* scenario, and was performed using an older version of L2 in which, as it turns out, the checkpointing functionality used by LPF was flawed. We consider a double fault occurring at 3301 seconds into the simulation. In this case an open microswitch, on the LOX vent/relief solenoid valve `sv31`, fails and `sv31` fails open. Here is the sequence of events:

1. a command `open` is issued to `sv31`,

2. a command `close` is issued to `sv31`,
3. the open microswitch of `sv31` breaks,
4. `sv31` fails in stuck-open position.

Technically, the faults in the two last items are injected by LPF into the simulator. Here is the same sequence as reported by LPF:

```
-- Transition:
{<Initial>}
-- Transition:
{command test.sv31.valveCmdIn=open}
-- Transition:
{command test.sv31.valveCmdIn=close}
-- Transition:
{fault test.sv31.openMs.mode=faulty}
-- Transition:
{fault test.sv31.sv.mode=stuckOpen}

-- Final Simulator state:
test.sv31.openMs.mode=faulty
test.sv31.sv.mode=stuckOpen
```

The microswitch failure is undetected (Livingstone still reports the empty candidate after step 3), but the stuck valve causes a fault diagnosis: after step 4, Livingstone reports the following candidates:

```
Candidate 0)
     5#test.vr01.modeTransition=stuckOpen :2
Candidate 1)
     -#test.vr01.modeTransition=stuckClosed :2
Candidate 2)
     -#test.sv31.modeTransition=stuckClosed :3
Candidate 3)
     -#test.vr01.modeTransition=stuckOpen :2
     3#test.vr01.modeTransition=stuckClosed :2
Candidate 4)
     5#test.forwardLo2.rp1sv.modeTransition=unknownFault :5
Candidate 5)
     -#test.forwardLo2.rp1sv.modeTransition=unknownFault :5
Candidate 6)
     3#test.forwardLo2.rp1sv.modeTransition=unknownFault :5
Candidate 7)
     5#test.sv31.modeTransition=stuckOpen :5
Candidate 8)
     -#test.sv03.sv.modeTransition=stuckClosed :3
     3#test.sv03.openMs.modeTransition=faulty :3
Candidate 9)
     -#test.sv03.sv.modeTransition=stuckClosed :3
     -#test.sv03.openMs.modeTransition=faulty :3
```

At this point, LPF reports a violation of the CM error condition, indicating that none of the Livingstone candidates match the modes from the simulator. The fault is detected (otherwise no faulty candidates would be generated), but incorrectly diagnosed (note that candidate 7 *subsumes*, but does not *match*, the actual faults). This data may suggest, for example, that Livingstone is not properly generating some valid candidates.

To further interpret and check the results reported by LPF, we ran `l2test` on the L2 scenario for diagnosis associated to this error. We obtained the following list of candidates, which differs from those obtained above from LPF:

```
L2> Step 6; 3529 variables; 0 conflicts before; 242 after.
CBFS: search found 5 candidate(s), more possible (searched 12)
The 8 candidates are:
Candidate 0)
     5#test.vr01.modeTransition=stuckOpen :2
Candidate 1)
     4#test.vr01.modeTransition=stuckOpen :2
Candidate 2)
     4#test.forwardLo2.modeTransition=unknownFault :5
Candidate 3)
     3#test.forwardLo2.modeTransition=unknownFault :5
Candidate 4)
     4#test.sv31.sv.modeTransition=stuckOpen :5
Candidate 5)
     5#test.forwardLo2.modeTransition=unknownFault :5
Candidate 6)
     -#test.forwardLo2.modeTransition=unknownFault :5
Candidate 7)
     5#test.sv31.sv.modeTransition=stuckOpen :5
```

Although none of these match the actual faults either (i.e. there is indeed a CM error in this state), the difference also shows that the results from LPF were flawed. Further analysis revealed that the discrepancy originated from a checkpointing bug, where Livingstone did not properly restore its internal constraint network when restoring checkpoints, resulting in a corrupted internal state and incorrect diagnosis results. With our help, the error was localized and fixed in a new release of the Livingstone program.

5.4 Example Scenario 2: SV02 stuck open

The second example considers one of the five errors reported using candidate subsumption on the Random scenario. It involves a solenoid valve, `sv02`, which sends pressurized helium into a rocket propellant tank. A command `close` is issued to the valve, but the valve fails and remains open—in LPF terms, a fault is injected in the simulator. This scenario corresponds to the following sequence of LPF events:

```
-- Transition:
{<Initial>}
-- Transition:
{command test.sv02.valveCmdIn=close}
-- Transition:
{fault test.sv02.rp1sv.mode=stuckOpen}

-- Final Simulator state:
test.sv02.rp1sv.mode=stuckOpen
```

The following sample output lists the candidates reported by Livingstone after the fault occurs in `sv02`:

```
 -- Transition:
{fault test.sv02.rp1sv.mode=stuckOpen}
   Candidate 0)
        4#test.sv02.openMs.modeTransition=faulty :3
   Candidate 1)
        3#test.sv02.openMs.modeTransition=faulty :3
   Candidate 2)
        2#test.sv02.openMs.modeTransition=faulty :3
   Candidate 3)
```

```
     -#test.sv02.openMs.modeTransition=faulty :3
Candidate 4)
     -#test.sv02.rp1sv.modeTransition=unknown :4
```

The injected fault, `test.sv02.rp1sv.mode=stuckOpen`, is detected (otherwise no faulty candidates would be generated) but incorrectly diagnosed: none of these candidates matches or subsumes the correct fault. The first four candidates consist of a faulty open microswitch sensor at different time steps (microswitches report the valve's position). The last candidate consists of an unknown fault mode. The L2 scenario corresponding to this error was replayed in `l2test` and produced the same results, confirming the validity of the results from LPF.

There are a number of explanations that could account for the improper diagnosis resulting in the above error condition violation. For example the following are all plausible explanations: the number of candidates returned or the Livingstone search space (number of candidates Livingstone searches over to find the diagnosis) may need to be increased; or issues regarding the Livingstone program itself are in need of deliberation. However, further analysis by application specialists revealed that fault ranks in the X-34 model needed re-tuning, which resolved the problem.

6 Conclusions and Perspectives

Currently more extensive testing and analysis is ongoing. Future experiments will include heuristic search generated results using different fitness functions on larger scenario files.

Livingstone PathFinder (LPF) is a software tool for automatically analyzing model-based diagnosis applications across a wide range of scenarios. LPF is under active development, in close collaboration with Livingstone application developers at NASA Ames. After considerable efforts resolving technical issues in both LPF and relevant parts of Livingstone, we are now returning useful results to application specialists, who in turn reciprocate much needed feedback and suggestions on further improvements. The *candidate subsumption* error condition is the latest fruit of this interaction. Directions for further work include new search strategies and heuristics, additional error conditions including capture of application-specific criteria, improved post-treatment and display of the large amount of data that is typically produced.

We are also investigating adapting LPF to use MIT's Titan model-based executive [1], which offers a more comprehensive diagnosis capability as well as a reactive controller. This extends the verification capabilities to involve the remediation actions taken by the controller when faults are diagnosed. In this regard, LPF can be considered as evolving towards a versatile system-level verification tool for model-based controllers.

Acknowledgments

This research is funded by NASA under ECS Project 2.2.1.1, *Validation of Model-Based IVHM Architectures*, and partly performed by QSS Group Inc. under CSRDS Task 091. The authors would like to thank Livingstone application developers at NASA Ames, especially Adam Sweet, Anupa Bajwa and Sandra Hayden for their active cooperation and Livingstone lead developer Lee Brownston for his responsive support.

References

[1] B. Williams, M. Ingham, S. Chung and P. Elliot, Model-based Programming of Intelligent Embedded Systems and Robotic Space Explorers. In *Proceedings of the IEEE Modelings and Design of Embedded Software Conference*, vol. 9, no.1 pp. 212-237, 2003.

[2] L. Fleming, T. Hatfield and J. Malin. Simulation-Based Test of Gas Transfer Control Software: CONFIG Model of Product Gas Transfer System. Automation, Robotics and Simulation Division Report, AR&SD-98-017, (Houston, TX:NASA Johnson Space Center Center, 1998).

[3] C. Meyer, H. Cannon, Propulsion IVHM Technology Experiment Overview, In *Proceedings of the IEEE Aerospace Conference*, (March 2003).

[4] B. Williams and P. Nayak, A Model-based Approach to Reactive Self-Configuring Systems. *Proceedings of the National Conference on Artificial Intelligence*, (August 1996).

[5] A. Bajwa and A. Sweet. The Livingstone Model of a Main Propulsion System. In *Proceedings of the IEEE Aerospace Conference*, (March 2003).